BRICS

Basic Research in Computer Science

BRICS RS-95-8 Hougaard et al.: Type Inference of Turbo Pascal

Type Inference of Turbo Pascal

Ole I. Hougaard
Michael I. Schwartzbach
Hosein Askari

BRICS Report Series RS-95-8

ISSN 0909-0878 February 1995

**Copyright © 1995, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@daimi.aau.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.aau.dk/BRICS/`
`ftp ftp.brics.aau.dk (cd pub/BRICS)`

Type Inference of Turbo Pascal

Ole I. Hougaard, Michael I. Schwartzbach, Hosein Askari
{hougaard,mis,hosask}@daimi.aau.dk
BRICS*
Computer Science Department
Aarhus University
8000 Aarhus C, Denmark

Abstract

Type inference is generally thought of as being an exclusive property of the functional programming paradigm. We argue that such a feature may be of significant benefit for also standard imperative languages. We present a working tool (available by WWW) providing these benefits for a full version of Turbo Pascal. *It has the form of a preprocessor that analyzes programs in which the type annotations are only partial or even absent. The resulting program has full type annotations, will be accepted by the standard* Turbo Pascal *compiler, and has polymorphic use of procedures resolved by means of code expansion.*

Keywords: imperative languages, type inference.

1 Explicit versus Implicit Typing

Several quite different programming paradigms are today proposed for general-purpose programming. Two notable candidates are the imperative languages (exemplified by Pascal) and the functional languages (exemplified by ML).

There has been much debate about the relative merits of these paradigms. Some arguments have an ideological flavor, while others focus on differences in the implementations of existing systems. In the latter category, functional languages have been criticized for being somewhat inefficient but commended for supporting simple yet powerful features, while imperative languages seem almost to be ascribed the dual properties.

*Basic Research in Computer Science, Center of the Danish Research Foundation

However, some of the attractive features of e.g. ML are not necessarily exclusive properties of the functional programming paradigm. For example, heap-allocated recursive data types could equally well be incorporated into a Pascal-like language [2]. In this paper we focus on another often cited advantage of modern functional languages, viz. automatic *type inference*, and we argue that such a feature may be of significant methodological benefit for also traditional imperative languages. We present an algorithm that allows type inference of general Pascal programs. Furthermore, this algorithm has been implemented for a full version of Turbo Pascal.

What is then the task of type inference? This is a well-defined problem for any language that supports type annotations and type checking. Normally, the programmer provides type annotations for all variables and the type checker then proceeds to verify that all the type constraints are respected. Type inference is the more difficult task of accepting a program in which the type annotations are only partial or even absent and then deciding if there exists a choice of type annotations with which the program would be accepted by the normal type checker. Generally, the annotated program should be presented as well. A separate notion is that of *polymorphism* which, as we shall see, is intimately connected with type inference.

Note that with type inference we do not fall back to *untyped* programs, which of course are subject to all sorts of damaging type errors. Rather, our programs are *implicitly* typed, as valid types must certainly exist even if they are not supplied by the programmer. The ambition behind type inference is to obtain all the safety benefits of typed programs while avoiding the problems with verbose and cumbersome annotations.

It can certainly be argued that there are further benefits to having programs be *explicitly* typed. After all, when the programmer states his intentions up front, then certain logical errors may be caught by the element of redundancy that separates type checking from type inference.

This aspect is clearly realized by proponents of type inference who offer several arguments in reply. First of all, comparing the inferred types to those that were intended provides a similar degree of redundancy. Secondly, type inference may discover that parts of your program are more general than you had originally realized, thus widening its applicability. And finally, if parameter types are only given implicitly, then a procedure may be given more than one type in different contexts—thus allowing polymorphism. All these benefits are major selling points for functional languages, and their potential applicability to imperative languages should be given serious consideration.

Detractors of e.g. Pascal could also interject that its type system, in particular the notion of type equivalence, makes it difficult for the programmer to correctly state his actual intentions in advance.

In any case, a language such as Pascal already performs a modicum of type inference. Only the types of variables must be given explicitly, and from this information the types of all expressions are inferred. For example, if x is declared to be of type Real, then the type of the expression x+1 is inferred as follows: the value of x is of type Real; there is a version of + with type Real×Integer → Real; the constant 1 has type Integer; thus we can conclude that x+1 has type Real. This sort of inference is perhaps too trivial to gain much notice, but it is there and forms a basis for using implicit types in larger parts of programs.

In present Pascal implementations the types of variables are given by explicit annotations. Our algorithm allows the possibility of supplying only some of these annotations. This yields several contributions: a proof-of-concept that type inference is possible for traditional imperative languages; a tool for Turbo Pascal that implements this algorithm and allows polymorphic procedures; and a platform for studying the methodological impact of implicit typing.

In the following sections we show how to generalize the techniques for type inference from unification to constraint solving; we sketch the constraint solver that is necessary for a Pascal type system; and we describe a concrete tool that has been built as a preprocessor for a version of Turbo Pascal.

2 Techniques for Type Inference

In functional languages as ML, type inference may be done by recursively going through the parse tree and assigning a type to each node. The type of a parse tree node can be derived from the types of the subexpressions. For example when we want to find the type of the expression $(\mathsf{e}_1\mathsf{e}_2)$, we first find the types of e_1 and e_2; let us call these τ_1 and τ_2 respectively. We know that the type of e_1 must be a function type, taking something of the type of e_2 as its argument. We can write this as the equation $\tau_1 = \tau_2 \rightarrow \alpha$, where α is a *type variable* corresponding to the return type of e_1 that can be instantiated with any type. In order to solve the equation we apply *unification* to the two sides of the equation. Unification finds a most general instantiation of type variables, so that the two types become equal. The type of $(\mathsf{e}_1\mathsf{e}_2)$ is simply the instantiation that the unification algorithm finds for α. This technique was used by Milner for type inference of ML in [5].

The above technique allowed type variables to stand for any type. By using type variables in this manner we can represent the set of all possible types for a parse tree node (see [1]). Thus the success of this approach relies on said representation and the fact that we could compute the representation of the solutions to the *constraint* $\tau_1 = \tau_2 \rightarrow \alpha$.

In the general case we cannot expect to find a proper representation and compute the solutions to the constraints within this representation. A more general technique is that of generating and solving a set of constraints for the specific program. In this case we will not try to derive the type from those of the subexpressions. Instead we generate type variables representing the (yet unknown) types of all parse tree nodes and further generate the appropriate constraints relating these type variables. In the case of the expression $(\mathsf{e}_1\mathsf{e}_2)$ we generate the constraint $[\![\mathsf{e}_1]\!] = [\![\mathsf{e}_2]\!] \rightarrow [\![(\mathsf{e}_1\mathsf{e}_2)]\!]$, where we use $[\![\mathsf{e}]\!]$ to stand for the type variable representing the type of expression e. Now we have reduced the problem of type inference to that of finding a solution to a set of constraints. In the case of ML we can again solve the constraints by a single application of the unification algorithm. Wand [6] has used this technique for type inference of the simply typed λ-calculus (ML without polymorphic let).

In Turbo Pascal we use this constraint technique. In ML we could limit ourselves to generating constraints of the form $[\![\mathsf{e}]\!] = [\![\mathsf{e}']\!] \rightarrow [\![\mathsf{e}'']\!]$, but Turbo Pascal has much more complex typing rules and we need a substantially richer class of constraints.

Consider for example the simple assignment, x:=e. The typing rules of Turbo Pascal demands that the type of e is assignment compatible to the type of x, that is, $[\![\mathsf{x}]\!] := [\![\mathsf{e}]\!]$ is the generated constraint, where we use := to denote the assignment compatibility relation. Similarly, we generate constraints of the form $[\![\mathsf{e}]\!]\ \mathrm{Tc}\ [\![\mathsf{e}']\!]$ when the types of e and e' should be type compatible; $\mathrm{Op}([\![\mathsf{e}]\!], [\![\mathsf{e}']\!], [\![\mathsf{e}'']\!])$ when the type of e'' should be the result type of a binary operation between e and e'; and $[\![\mathsf{e}]\!]\ \mathrm{Io}\ [\![\mathsf{e}']\!]$ when e' should be writable to a file which has the type of e.

For example, we generate the constraint $\mathrm{Op}([\![\mathsf{e}]\!], [\![\mathsf{e}']\!], [\![\mathsf{e+e}']\!])$ for the expression e+e' and the constraint $[\![\mathsf{f}]\!]\ \mathrm{Io}\ [\![\mathsf{e}]\!]$ for the statement **write**(f,e).

The expression e−e' does not apply to strings as opposed to e+e'. Hence the constraint $\mathrm{Op}([\![\mathsf{e}]\!], [\![\mathsf{e}']\!], [\![\mathsf{e-e}']\!])$ is too liberal. We restrict it by imposing further constraints on the types of e, e' and e−e', namely $[\![\mathsf{e}]\!], [\![\mathsf{e}']\!], [\![\mathsf{e-e}']\!] \in \mathcal{M}_-$, where $\mathcal{M}_-$ is the set of types on which '−' can operate.

As a further example of the use of constraints of the form $[\![\mathsf{e}]\!] \in M$ we can regard a **for**-statement. Among the constraints generated for the statement:

for x := e **to** e' **do** S

we have $[\![\mathsf{e}]\!], [\![\mathsf{e}']\!] \in \mathcal{O}$, where $\mathcal{O}$ is the set of *ordinal types*.

In connection with structured types we get the constraints $\mathrm{Rec}_\alpha([\![\mathsf{e}]\!], [\![\mathsf{e}']\!])$ requiring that $[\![\mathsf{e}]\!]$ is a record with a field α that has type $[\![\mathsf{e}']\!]$, and $[\![\mathsf{e}]\!] = T([\![\mathsf{e}_1]\!], \ldots, [\![\mathsf{e}_n]\!])$, where T is a type constructor. For example, we get the constraint $\mathrm{Rec}_\alpha([\![\mathsf{x}]\!], [\![\mathsf{x}.\alpha]\!])$ for the expression x.α and $[\![\mathsf{x}]\!] = \,\hat{}\,[\![\mathsf{x}\hat{}\,]\!]$ for the expression x^.

Finally, we have the simple constraint $[\![\mathsf{e}]\!] = [\![\mathsf{e}']\!]$ in connection with variable parameters, where the actual type must equal the formal type, and expressions like −e, where we have the constraint $[\![-\mathsf{e}]\!] = [\![\mathsf{e}]\!]$.

All in all, we have the following kinds of constraints:

- $[\![\mathsf{e}]\!] \in \mathcal{M}$, where $\mathcal{M}$ is from a fixed finite set of sets of types.
- $[\![\mathsf{e}]\!] = T([\![\mathsf{e}_1]\!], \ldots, [\![\mathsf{e}_n]\!])$, where T is a type constructor.
- $\mathrm{Rec}_\alpha([\![\mathsf{e}]\!], [\![\mathsf{e}']\!])$

- $[\![\mathsf{e}]\!] = [\![\mathsf{e'}]\!]$
- $[\![\mathsf{e}]\!]$ Tc $[\![\mathsf{e'}]\!]$, $[\![\mathsf{e}]\!] := [\![\mathsf{e'}]\!]$, $\mathrm{Op}([\![\mathsf{e}]\!], [\![\mathsf{e'}]\!], [\![\mathsf{e''}]\!])$, and $[\![\mathsf{e}]\!]$ Io $[\![\mathsf{e'}]\!]$

Example: Consider the following recursive function for computing the factorial of a number.

```
Function fac(n: Integer): Integer;
begin
    if n=0 then
      fac := 0
    else
      fac := n*fac(n-1)
end
```

We generate the following set of constraints for the function:

$$
\begin{array}{l}
[\![\mathsf{1}]\!] \in \mathcal{I} \\
[\![\mathsf{0}]\!] \in \mathcal{I} \\
[\![\mathsf{n}]\!] = \mathsf{Integer} \\
[\![\mathsf{fac}]\!] = \mathsf{Integer} \\
[\![\mathsf{n{=}0}]\!] = \mathsf{Boolean} \\
[\![\mathsf{n}]\!] \ \mathrm{Tc}\ [\![\mathsf{0}]\!] \\
[\![\mathsf{n}]\!], [\![\mathsf{0}]\!] \in \mathcal{M}_{=} \\
[\![\mathsf{fac}]\!] := [\![\mathsf{1}]\!] \\
[\![\mathsf{fac}]\!] := [\![\mathsf{n*fac(n-1)}]\!] \\
\mathrm{Op}([\![\mathsf{n}]\!], [\![\mathsf{fac(n-1)}]\!], [\![\mathsf{n*fac(n-1)}]\!]) \\
[\![\mathsf{n}]\!], [\![\mathsf{fac(n-1)}]\!], [\![\mathsf{n*fac(n-1)}]\!] \in \mathcal{M}_{*} \\
[\![\mathsf{fac(n-1)}]\!] = \mathsf{Integer} \\
\mathrm{Op}([\![\mathsf{n}]\!], [\![\mathsf{1}]\!], [\![\mathsf{n-1}]\!]) \\
[\![\mathsf{n}]\!], [\![1]\!], [\![\mathsf{n-1}]\!] \in \mathcal{M}_{-}
\end{array}
$$

where $\mathcal{I} \subset \mathcal{O}$ is the set of integer types including all subranges of integers. The following is a solution to the set of constraints:

$$
\begin{array}{rcl}
[\![\mathsf{n}]\!] & = & \mathsf{Integer} \\
[\![\mathsf{0}]\!] & = & \mathsf{Integer} \\
[\![\mathsf{n{=}0}]\!] & = & \mathsf{Boolean} \\
[\![\mathsf{fac}]\!] & = & \mathsf{Integer} \\
[\![\mathsf{1}]\!] & = & \mathsf{Integer} \\
[\![\mathsf{n*fac(n-1)}]\!] & = & \mathsf{Integer} \\
[\![\mathsf{fac(n-1)}]\!] & = & \mathsf{Integer} \\
[\![\mathsf{n-1}]\!] & = & \mathsf{Integer}
\end{array}
$$

□

Note that even in the case of a program with explicit type annotation, we must infer some of the types. In the above example we inferred the type Integer for the expression n*fac(n-1). When the type annotation is present, we can use the technique of deriving the possible types of a parse tree node from the possible types of the subexpressions. This technique is widely used in type checkers. When some or all of the type annotations are missing, we have to use the technique of generating and solving constraints.

Example: Consider the implicit version of the above example function:

```
Function fac(n);
begin
    if n=0 then
        fac := 0
    else
        fac := n*fac(n-1)
end
```

The set of constraints has no direct information about the types of n or fac:

$$
\begin{array}{l}
[\![\mathsf{1}]\!] \in \mathcal{I} \\
[\![\mathsf{0}]\!] \in \mathcal{I} \\
[\![\mathsf{n=0}]\!] = \mathsf{Boolean} \\
[\![\mathsf{n}]\!] \ \mathrm{Tc} \ [\![\mathsf{0}]\!] \\
[\![\mathsf{n}]\!], [\![\mathsf{0}]\!] \in \mathcal{M}_{=} \\
[\![\mathsf{fac}]\!] := [\![\mathsf{1}]\!] \\
[\![\mathsf{fac}]\!] := [\![\mathsf{n*fac(n-1)}]\!] \\
\mathrm{Op}([\![\mathsf{n}]\!], [\![\mathsf{fac(n-1)}]\!], [\![\mathsf{n*fac(n-1)}]\!]) \\
[\![\mathsf{n}]\!], [\![\mathsf{fac(n-1)}]\!], [\![\mathsf{n*fac(n-1)}]\!] \in \mathcal{M}_{*} \\
[\![\mathsf{fac(n-1)}]\!] = \mathsf{Integer} \\
\mathrm{Op}([\![\mathsf{n}]\!], [\![\mathsf{1}]\!], [\![\mathsf{n-1}]\!]) \\
[\![\mathsf{n}]\!], [\![\mathsf{1}]\!], [\![\mathsf{n-1}]\!] \in \mathcal{M}_{-}
\end{array}
$$

The solution from the previous example is a solution here, too; but there is also a very different solution:

$$\begin{aligned}
[\![\mathsf{n}]\!] &= \mathsf{Real}\\
[\![\mathsf{0}]\!] &= \mathsf{Integer}\\
[\![\mathsf{n{=}0}]\!] &= \mathsf{Boolean}\\
[\![\mathsf{fac}]\!] &= \mathsf{Real}\\
[\![\mathsf{1}]\!] &= \mathsf{Integer}\\
[\![\mathsf{n{*}fac(n{-}1)}]\!] &= \mathsf{Real}\\
[\![\mathsf{fac(n{-}1)}]\!] &= \mathsf{Real}\\
[\![\mathsf{n{-}1}]\!] &= \mathsf{Real}
\end{aligned}$$

□

Since all the type rules of Turbo Pascal can be expressed in this manner, we have reduced the problem of type inference to that of finding a solution to a set of certain kinds of constraints. We now have to exhibit an algorithm that solves such a set of constraints.

3 An Algorithm for Turbo Pascal

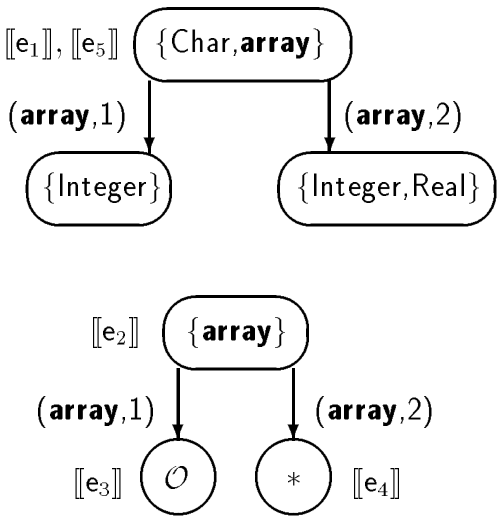
In the algorithm we will maintain a data structure containing all the solutions to an increasing set of constraints generated dynamically from the program. This just leaves the task of selecting one of these solutions from the data structure. Selecting a solution from the data structure we will present here is in general a non-trivial task, but in Turbo Pascal we can use some specific features of the type system to obtain a solution quite easily.

We represent the set of solutions by a graph, where the nodes contain a set of basic types and type constructors, e.g. a node could contain the type Integer or the constructor **array**. We shall use the term *label* as a common name for basic types and type constructors. For each constructor there are edges to the nodes representing its component types. We let every type variable point to the node representing its type. It is possible for several type variables to point to the same node.

Example: Assume that we have the type variables $[\![\mathsf{e}_1]\!], [\![\mathsf{e}_2]\!], \ldots, [\![\mathsf{e}_5]\!]$, and the constraints (possibly not from any real Turbo Pascal program):

$$\begin{aligned}
[\![\mathsf{e}_1]\!] &= [\![\mathsf{e}_5]\!]\\
[\![\mathsf{e}_5]\!] &\in \{\mathsf{Integer}, \mathsf{Char}, \textbf{array}\ \mathsf{Integer}\ \textbf{of}\ \mathsf{Real}, \textbf{array}\ \mathsf{Integer}\ \textbf{of}\ \mathsf{Integer}\}\\
[\![\mathsf{e}_1]\!] &\in \{\mathsf{Char}, \textbf{array}\ \mathsf{Integer}\ \textbf{of}\ \mathsf{Real}, \textbf{array}\ \mathsf{Integer}\ \textbf{of}\ \mathsf{Integer},\\
&\qquad \textbf{array}\ \mathsf{Boolean}\ \textbf{of}\ \mathsf{Integer}\}\\
[\![\mathsf{e}_2]\!] &= \textbf{array}\ [\![\mathsf{e}_3]\!]\ \textbf{of}\ [\![\mathsf{e}_4]\!]
\end{aligned}$$

We can represent the set of all solutions to the above constraints as follows:



The label '$*$' stands for any Turbo Pascal type. Note that the set of possible types for $[\![e_1]\!]$ and $[\![e_5]\!]$ is found through the intersection of the set of possible types for $[\![e_1]\!]$ and the set of possible types for $[\![e_5]\!]$. Note also that the representation incorporates the extra constraint that the index type of an **array**-type must be an ordinal type.
□

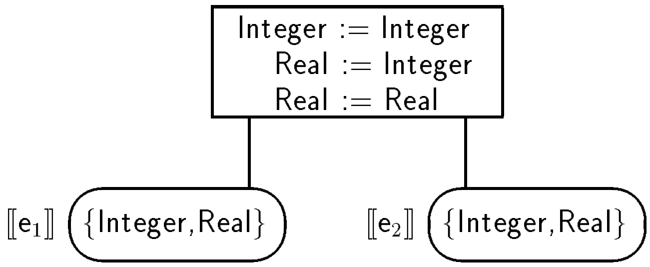
An arbitrary solution can be found simply by choosing a label for each node. In the above example the types of $[\![e_1]\!]$ and $[\![e_5]\!]$ are guaranteed to be equal, since they point to same node. This represents the Turbo Pascal notion of *name equivalence*.

In the above example it was possible to choose a type for each node independently of the types chosen for other nodes. In the presence of complicated constraints like $[\![x]\!] := [\![e]\!]$, this is not the case. Assume for example that $[\![x]\!]$ and $[\![e]\!]$ both point to (different) nodes containing the set {Integer, Real}. If $[\![x]\!]$ is a Real, then $[\![e]\!]$ can be either a Real or an Integer, but if $[\![x]\!]$ is an Integer, then $[\![e]\!]$ will have to be an Integer as well. We represent this by introducing relations between nodes in the graph.

Example: Consider the constraints:

$$\begin{array}{lcl} [\![e_1]\!] & \in & \{\mathsf{Integer}, \mathsf{Real}\} \\ [\![e_2]\!] & \in & \{\mathsf{Integer}, \mathsf{Real}\} \\ [\![e_1]\!] & := & [\![e_2]\!] \end{array}$$

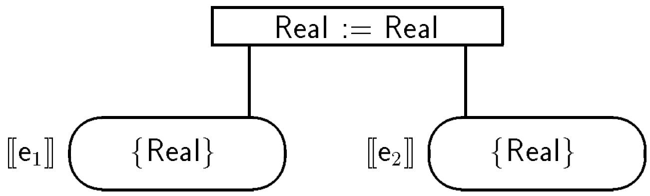
We represent these as:



The box is representing the relation :=. Each line of the box represents a possible solution to the relation. We use this representation of the relation to *reduce* the graph in the following way. Suppose we read the constraint:

$$[\![\mathsf{e}_2]\!] = \mathsf{Real}$$

Now we can reduce the set of labels for the right node to $\{\mathsf{Real}\}$. Looking at the box we can see that the possibilities Integer := Integer and Real := Integer are no longer relevant. We can thus remove them, and the only remaining possibility is Real := Real. We get the following representation:



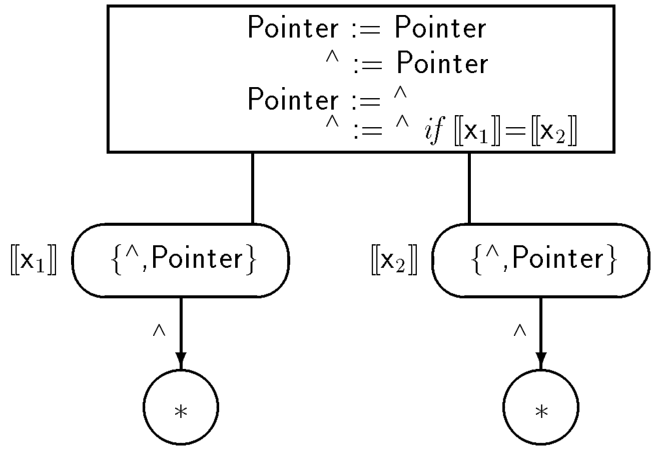
□

There are four different relations corresponding to the constraints $[\![\mathsf{e}]\!]$ Tc $[\![\mathsf{e'}]\!]$, $[\![\mathsf{e}]\!] := [\![\mathsf{e'}]\!]$, $\mathrm{Op}([\![\mathsf{e}]\!], [\![\mathsf{e'}]\!], [\![\mathsf{e''}]\!])$, and $[\![\mathsf{e}]\!]$ Io $[\![\mathsf{e'}]\!]$. In Turbo Pascal these relations have arity 2 or 3, but in general we may consider relations of a greater arity as well. The relations are adjusted to the sets of labels in the corresponding nodes, so that only the relevant cases are present. A further aspect of these constraints can be seen in an example using pointers.

Example: Consider the constraints:

$$\begin{array}{lcl} [\![\mathsf{x}_1]\!] & \in & \{\mathsf{Pointer}\} \cup \{\hat{}(T) \mid T \text{ is a type}\} \\ [\![\mathsf{x}_2]\!] & \in & \{\mathsf{Pointer}\} \cup \{\hat{}(T) \mid T \text{ is a type}\} \\ [\![\mathsf{x}_1]\!] & := & [\![\mathsf{x}_2]\!] \end{array}$$

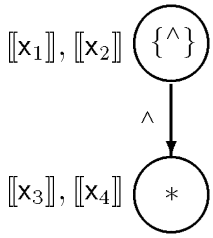
They give rise to the following representation:



Note the fourth line in the box: $\wedge := \wedge$ *if* $[\![x_1]\!] = [\![x_2]\!]$. It is a *conditional possibility*. It says, that if x_1 and x_2 are both pointers to some other type, then they have to be name equivalent in order to be assignment compatible. If we read the constraints:

$$
\begin{aligned}
[\![x_1]\!] &= \wedge[\![x_3]\!] \\
[\![x_2]\!] &= \wedge[\![x_4]\!]
\end{aligned}
$$

in the above situation we get that the only remaining possibility is the conditional $\wedge := \wedge$ *if* $[\![x_1]\!] = [\![x_2]\!]$. We will thus identify the nodes, and we get the following representation.



Note how this *collapsing* of nodes results in a subsequent collapsing of their successor nodes.
□

There are three operations on graphs as the above.

- *Collapsing two nodes.* The labels of the collapsed node are intersections of the labels of the original nodes. The relations connected to the nodes are transformed accordingly (their arities may decrease). This operation may result in subsequent collapsings of successor nodes and reductions of relations.

- *Reducing a relation.* Remove all cases that are no longer possible. May result in collapsing and reduction of nodes.

- *Reducing a node.* Remove one or more labels from the set of labels contained in the node. May result in reduction of relations.

As it is seen, the operations are mutually dependent (i.e. recursive). The termination of the operations is ensured by the fact that all operations reduce the size of the graph. With a suitable representation of the graph the operations can be implemented in such a way that the total time of any sequence of operations is $O(n^2 \log n)$, where n is the size of the Turbo Pascal program.

The remaining task is that of choosing one of the solutions represented in the graph. This could be a nontrivial task because the relations may not "fit". That is, there may not be a global solution corresponding to a certain case in a particular relation. Similarly, even if a node contains a certain label, there is no guarantee that the node has this label in any global solution. The problem is that the local information can collide with demands in other parts of the graph. In general, finding whether a solution exists is NP-complete.

We must examine more closely the relations generated for a Turbo Pascal program. Consider a node which contains the label Boolean. This node could be attached to a number of relations, Tc being one of them. In Turbo Pascal, Boolean is only type compatible to itself, so the node in the other end of the Tc relation must contain Boolean as well. If we have a solution to this particular relation, then both nodes have type Boolean which we can safely choose. Similar arguments apply to the other relations.

Assume now that a particular node contains not Boolean but rather Real. Consider the case where the node represents the return type of a binary operation, for example +. Then either the two other attached nodes will both contain Real or one will contain Real and the other will contain Integer. In the former case we can safely choose Real as a type for all the nodes. In the latter there will be a solution in which the two types that contain Real have type Real, and the one that does not has type Integer. So we can apply a strategy of always preferring Real if there is no Boolean, and Integer if there is neither Boolean nor Real. Note that if the node of the return type did not contain Integer, then there will be no solution where the two other nodes both have type Integer. Thus we cannot interchange Integer and Real in this strategy.

The above represents an ordering between the labels. Boolean is the least, then comes Real followed by Integer. By looking at all the labels we can see that there exists a general ordering of all the labels, such that it is a sound strategy always to pick the least of the labels contained in a node. Thus it is straightforward to find a solution from the graph in the case of Turbo Pascal. The total time for the type inference of a program of size n remains at $O(n^2 \log n)$.

4 The Turbo Pascal Tool

The Turbo Pascal programming language comes equipped with a fast and efficient compiler. Hence there is no need to reinvent the wheel by writing a new compiler. Instead we have implemented the Turbo Pascal tool as a preprocessor, which accepts a Turbo Pascal program without or partially without type annotations and returns a typed program that can in turn be compiled by the Turbo Pascal compiler. This means that when given the implicit version of the factorial function (as part of a whole program) it returns the following explicit version:

```
Type T1 = Real;

Function fac(n: T1): T1;
begin
    if n=0 then
      fac := 0
    else
      fac := n*fac(n-1)
end
```

Note the introduction of a new type identifier. The tool will always do this in order to get the name equivalence right. The tool is of course able to handle recursive types as in the following example:

```
Var z;

begin
    z^.a := z
end.
```

The resulting program is this:

```
Type T1 = ^T2;
     T2 = record
              a: T1;
          end;

Var z: T1;

begin
    z^.a := z
end.
```

As in the explicitly typed version of Turbo Pascal we must demand that recursive types contain a pointer type. If the '^' is left out in the above example the tool will respond with an error message.

Both of the above examples are reasonably small and simple. The tool handles them using these resources:

	factorial	recursive
# constraints	24	3
graph size	70	28
# type vars	12	3

The graph size is the largest number of nodes through the running of the program.

In the initial phase of a programming task it is usually unclear how the program will develop. In such cases it is not easy to provide the actual types that the program must use. Nevertheless this information must be provided in advance when programming in an explicitly typed language. By using type inference the programmer gets the freedom not to choose the actual types in advance, but instead concentrate on the algorithmic development of the program. Type inference is also a way to avoid cluttering up the program with a lot of redundant information.

However there are situations where it might be preferable to annotate some of the variables. In the above factorial example the inferred type for the variable n is Real. But this is not the preferred type. The call factorial(3.7) would never terminate anyway, and thus we might want to add the type annotation n: Integer to make sure that the factorial function is always called with an integer number. The tool is able to handle the extra information from type annotations.

One of the acclaimed features of ML and other functional languages is that of polymorphism. In a language with polymorphic procedures and functions, you can use the same function or procedure on arguments of different types. An often used example of this is the polymorphic length function. The polymorphic length function has type α list $\rightarrow$ Integer. The free type variable α can be instantiated with any type. This means that the length function can be applied to lists with any element type. This makes perfect sense because the length of a list has nothing to do with the elements of that lists. In ML polymorphism is inherent in the type system by allowing free type variables in the types, but it can also be seen as a consequence of the type inference. Take as an example the following code without type annotations:

```
Procedure double(Var x; y);
begin
    x := y+y
end;

Var a; b;
```

```
begin
    double(a,4);
    double(b,'cat')
end.
```

The constraints for the procedure double have among others the following solutions: $[\![x]\!] = [\![y]\!] = \mathsf{Real}$ and $[\![x]\!] = [\![y]\!] = \mathsf{String}$. This means that both of the calls of the function are type correct when seen in isolation. Unfortunately, we have to annotate the procedure with a single type for both x and y. This is impossible, since the types being forced on x and y from the two calls are completely incomparable. Instead we can lift the restriction that the types should be equal in the two cases by copying the procedure:

```
Type T1 = Real;
     T2 = String;

Procedure double(Var x: T1; y: T1);
begin
    x := y+y
end;

Procedure double1(Var x: T2; y: T2);
begin
    x := y+y
end;

Var a: T1; b: T2;

begin
    double(a,4);
    double1(b,'cat')
end.
```

The above example might suggest that a good strategy for implementing polymorphism would be simply to repeat copying procedures (and functions) until no two procedure calls are to the same procedure. This strategy, however, has two major deficiencies:

- In the presence of recursive procedures it will lead to infinite copying.
- Even when there are no recursive procedures it is inefficient.

The first problem can be solved simply by ignoring recursive calls and by always copying mutually recursive procedures together. To solve the second problem the preprocessor applies another strategy: It will not copy any procedure unless it is apparent from the constraints that the call of the procedure will lead to a type error. It will repeat this copying until it can find no more procedures which must be copied. Then no further copying will take place. Furthermore, before copying the preprocessor checks whether there is another copy of the procedure that can safely be called instead of the original. Thus the preprocessor employs a strategy of *conservative* copying. That is, the total size of the expanded program is no larger than an explicitly typed monomorphic program will have to be. However, when employing conservative copying we lose the full generality of polymorphism. In a realistic program, though, it is not likely that the type inference will fail due to insufficient copying.

As a more interesting example of the use of polymorphism see the following implicitly typed implementation of a polymorphic stack:

```
Procedure push(Var s; e);
     Var temp;
begin
     new(temp);
     temp^.next := s;
     s := temp;
     s^.elm := e;
end;

Function pop(Var x);
     Var temp;
begin
     temp := s;
     s := s^.next;
     pop := temp^.elm;
end;

Var a;
     b;
     c;
     d;

begin
   push(a, 94);
   push(b, true);
   c := pop(a);
   d := pop(b);
end.
```

There is no assumption about the types of the elements of the stack in the procedures push and pop, so the implementation should work for stacks of any type. For example Real and Boolean as in the example. The tool recognizes that the two calls to push have different types, and consequently makes a copy of push. This will give the two stacks a and b different types and thus result in a subsequent copying of the function pop. The result is as follows:

```
Type T1 = ^T5;
     T2 = ^T6;
     T3 = Real;
     T4 = Boolean;
     T5 = record
            elm: T3;
            next: T1;
          end;
     T6 = record
            elm: T4;
            next: T2;
          end;

Procedure push(Var s: T1; e: T3);
    Var temp;
begin
    new(temp);
    temp^.next := s;
    s := temp;
    s^.elm := e;
end;

Procedure push1(Var s: T2; e: T4);
    Var temp;
begin
    new(temp);
    temp^.next := s;
    s := temp;
    s^.elm := e;
end;
```

```
Function pop(Var x: T1): T3;
      Var temp;
begin
      temp := s;
      s := s^.next;
      pop := temp^.elm;
end;

Function pop1(Var x: T2): T4;
      Var temp;
begin
      temp := s;
      s := s^.next;
      pop := temp^.elm;
end;

Var a: T1;
      b: T2;
      c: T3;
      d: T4;

begin
   push(a, 94);
   push1(b, true);
   c := pop(a);
   d := pop1(b);
end.
```

Note that the inferred types are recursive.

The polymorphic examples are larger than the monomorphic ones. Especially the stack requires a large graph:

	double	stack
# constraints	14	50
graph size	77	192
# type vars	10	42

The obvious advantage with polymorphism is the compactness of the written code. Letting the tool do the necessary copying saves a lot of the programmers time. Who has not sighed with exasperation when programming the 17th identical version of linear search for the 17th version of a linked list? Furthermore, since the tool only performs the necessary copying it might in some cases find a solution having procedures than the one the programmer would come up with.

Even though the tool makes as few copies as possible, there are still examples in which the resulting program becomes very large. As an example take the following program:

```
Procedure P_{n+1}(x);
begin
end;

Procedure P_n(x);
    Var y; z;
begin
    y.a := x;
    z.a := x;
    y.b := 7;
    z.b := false;
    P_{n+1}(y);
    P_{n+1}(z)
end;
    ⋮
Procedure P_1(x);
    Var y; z;
begin
    y.a := x;
    z.a := x;
    y.b := 7;
    z.b := false;
    P_2(y);
    P_2(z)
end;

begin
    P_1(0)
end.
```

The expanded version of this program will contain $2^{n+1} - 1$ procedures. In ML there are similar problems with types of exponential size [3, 4]. In both cases it can be argued that the examples are highly artificial and that we will not encounter this behavior in practice.

A fully implemented prototype of the Turbo Pascal tool can be found at http://www.daimi.aau.dk/~hougaard/itp. This site contains the source code written in Turbo Pascal, together with the 220K binary executable file and some example files.

5 Conclusions and Future Work

The main result of this paper is that we have shown it to be possible to implement automatic type inference and polymorphism for a standard imperative language. This emphasizes that we should not always prefer one programming paradigm to another just on the grounds of the implemented features. In our case, one of the key advantages from ML has been lifted into a new context.

By implementing the tool for Turbo Pascal we have provided the means for experimenting and studying the impact of these new features on programming methodology. The examples shown here and the experiences with type inference and polymorphism in other programming languages certainly suggest that they are useful features. Especially so in a development fase where the lack of type annotations allows for greater flexibility.

The tool currently handles only the basic constructs from Turbo Pascal. Some of the more advanced features have not yet been implemented. Among the features that need to be considered is Turbo Pascal's *units*. The most straightforward way of handling units would be to apply type inference only inside of the unit. Once we have obtained a correct typing of the interface we will leave the interface as it is, thus demanding that all subsequent changes to the implementation of the unit respects this interface. In this way we can still provide separate compilation and use the built-in compiler. If we want separate compilation along with polymorphic units we will have to write a new compiler that can generate polymorphic code.

References

[1] Luis Damas and Robin Milner. Principal type schemes for functional programming. In *9th ACM conf. on Principels Of Programming Languages*, 1982.

[2] C.A.R. Hoare. Recursive data structures. *International Journal of Computer and Information Sciences*, 4:2:105–132, 1975.

[3] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *Journal of the ACM*, 41:368–398, 1994.

[4] Harry G. Mairson. Decidability of ML typing is complete for deterministic exponential time. In *Seventeenth Symposium on Principles of Programming Languages*, pages 382–401. ACM Press, January 1990.

[5] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348 – 375, 1978.

[6] M. Wand. A simple algorithm and proof for type inference. *Fundamentae Informaticae*, X:115 – 122, 1987.

Recent Publications in the BRICS Report Series

RS-95-8 **Ole I. Hougaard, Michael I. Schwartzbach, and Hosein Askari.** ***Type Inference of Turbo Pascal*****. February 1995. 19 pp.**

RS-95-7 **David A. Basin and Nils Klarlund.** ***Hardware Verification using Monadic Second-Order Logic*****. January 1995. 13 pp.**

RS-95-6 **Igor Walukiewicz.** ***A Complete Deductive System for the μ-Calculus*****. January 1995. 39 pp.**

RS-95-5 **Luca Aceto and Anna Ingólfsdóttir.** ***A Complete Equational Axiomatization for Prefix Iteration with Silent Steps*****. January 1995. 27 pp.**

RS-95-4 **Mogens Nielsen and Glynn Winskel.** ***Petri Nets and Bisimulations*****. January 1995. 36 pp. To appear in TCS.**

RS-95-3 **Anna Ingólfsdóttir.** ***A Semantic Theory for Value–Passing Processes, Late Approach, Part I: A Denotational Model and Its Complete Axiomatization*****. January 1995. 37 pp.**

RS-95-2 **François Laroussinie, Kim G. Larsen, and Carsten Weise.** ***From Timed Automata to Logic - and Back*****. January 1995. 21 pp.**

RS-95-1 **Gudmund Skovbjerg Frandsen, Thore Husfeldt, Peter Bro Miltersen, Theis Rauhe, and Søren Skyum.** ***Dynamic Algorithms for the Dyck Languages*****. January 1995. 21 pp.**

RS-94-48 **Jens Chr. Godskesen and Kim G. Larsen.** ***Synthesizing Distinguishing Formulae for Real Time Systems*****. December 1994. 21 pp.**

RS-94-47 **Kim G. Larsen, Bernhard Steffen, and Carsten Weise.** ***A Constraint Oriented Proof Methodology based on Modal Transition Systems*****. December 1994. 13 pp.**

RS-94-46 **Amos Beimel, Anna Gál, and Mike Paterson.** ***Lower Bounds for Monotone Span Programs*****. December 1994. 14 pp.**